[redacted]

Math

9

Report No. UIUCDCS-R-73-612   NSF - OCA - GJ-328 - 000001

DATA STRUCTURES AND OPERATOR FOR NEW ARRAY TYPES IN OL/2

by

John Leonard Larson

December 1973

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-73-612

DATA STRUCTURES AND OPERATOR FOR NEW ARRAY TYPES IN OL/2

by

John Leonard Larson

December 1973

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF-GJ-328 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, February 1974.

ACKNOWLEDGMENT

I wish to sincerely thank Professor J. Richard Phillips, who originated the OL/2 language, for his suggestion of this thesis topic, and for his advice and guidance toward its completion. I also wish to thank Mrs. Vivian Alsip, who did a fine job of typing this report, and to the National Science Foundation and the Department of Computer Science for their support.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The purpose of this thesis is to lay the theoretical background for the addition of new array types to the OL/2 language. The next section concerns itself with data structures for describing, and with algorithms for partitioning, Hessenburg arrays and their subarrays. The current language structure is adequate to handle this addition. It is not adequate, however, to handle band arrays and their subarrays, the subject of section 3. Investigations are made to find what additional information is necessary for describing these arrays and new data structures are proposed. New algorithms for partitioning are also derived. Finally, a rotate operator, which can generate new array types from the existing ones, is proposed and suggested implementations are given.

The design philosophy and present state of the OL/2 language are given in the references [1,2,3].

2. HESSENBURG FORM

2.1 OL/2 Background

Arrays in OL/2 are stored sequentially in row major order. Only those elements which are not theoretically zero are retained. Elements of an array are accessed by means of control information kept in the array control block, (ACB), corresponding to that array [2,3]. See Figure 1.

Of greatest importance in locating elements of an array are the ω, δR, and δD fields. The value of ω specifies the location of the first accessible element in that array. The value of δD is defined as the number of accessible elements in row i+1 minus the number of accessible elements in row i. For each array type this value is constant for all i. The following equation serves to define δR:

$$\delta R = \mathrm{LOC}(A(2,j)) - \mathrm{LOC}(A(1,j))$$

for any j provided both A(2,j) and A(1,j) are stored elements. For diagonal and strictly lower triangular arrays, δR is defined to be zero.

The location of an arbitrary array element can be found by:

$$\mathrm{LOC}(A(i+\delta i,j+\delta j)) = \omega + \delta R(i)\delta i + \delta D(\delta i(\delta i-1))/2 + \delta j$$

where $\omega = \mathrm{LOC}(A(i,j))$, $\delta R(i) = \delta R + (i-1)\delta D$, and $(i,j) \in \{(1,1),(1,2),(2,1)\}$ depending on the array type.

When an array is partitioned, the ρ field of the ACB of that array is set to point at a partition control block, (PCB). See Figure 2. The PCB contains information on how the array is partitioned and how the contents of the ACBs of the subarrays are to be filled in.

2.2 Upper and Lower Hessenburg Arrays

An Upper Hessenburg array, (UH), may be described as an n×n array

η	Δ
τ	δR
ω	δD
λ_1	μ_1
λ_2	μ_2
ρ	

where η - name of array

Δ - dimensionality

τ - geometric type

ω - origin

δR - row increment

δD - diagonal increment

λ_i - lower bounds

μ_i - upper bounds

ρ - partition pointer

Figure 1. Array control block

π	
r_0	c_0
r_1	$\vdots$
$\vdots$	c_{m+1}
r_{n+1}	ν
σ	

where π - array of 'part of' pointers

r_i - row partition lines

c_i - column partition lines

ν - number of partition lines

σ - subarray ACB pointers

Figure 2. Partition control block

where the subscripts of the theoretically non-zero elements satisfy the relation:

$$-(n-1) \leq i-j \leq 1 \quad \text{for } 1 \leq i,j \leq n.$$

For a Lower Hessenburg array, (LH), the relation is

$$-(n-1) \leq j-i \leq 1 \quad \text{for } 1 \leq i,j \leq n.$$

The number of non-zero elements in either array is $(n(n+3)-2)/2$.

The ACBs which describe Hessenburg arrays follow the same format as ACBs for other arrays in OL/2. The δR, δD, and ω fields for upper Hessenburg arrays are n, -1, and 1, respectively, where n is the order of the array. The corresponding ACB fields for a lower Hessenburg array are 2, 1, and 1. See Figure 3.

2.3 Partitioning of Hessenburg Arrays

Following the convention of triangular array partitioning, the partitioning of Hessenburg arrays is restricted to simultaneous identical row and column partition lines. Such partitioning yields the new array types upper right corner, (URC), and lower left corner, (LLC). See Figure 4. The types of the subarrays for arbitrary number of partition lines are given in the following table:

subarray	type
A<i,i>	same as parent
A<i,j>	RECT for j>i and UH parent j<i and LH parent
A<i,j>	URC for i=j+1 and UH parent LLC for j=i+1 and LH parent
other	null

1	2	3	4	5
6	7	8	9	10
	11	12	13	14
		15	16	17
			18	19

Upper Hessenburg order(5)

ACB

η = name	Δ = 2
τ = UH	δR= 5
ω = 1	δD= -1
λ_1= 1	μ_1= 5
λ_2= 1	μ_2= 5
ρ = null	

1	2			
3	4	5		
6	7	8	9	
10	11	12	13	14
15	16	17	18	19

Lower Hessenburg order(5)

ACB

η = name	Δ = 2
τ = LH	δR= 2
ω = 1	δD= 1
λ_1= 1	μ_1= 5
λ_2= 1	μ_2= 5
ρ = null	

Figure 3. Hessenburg arrays and corresponding ACBs

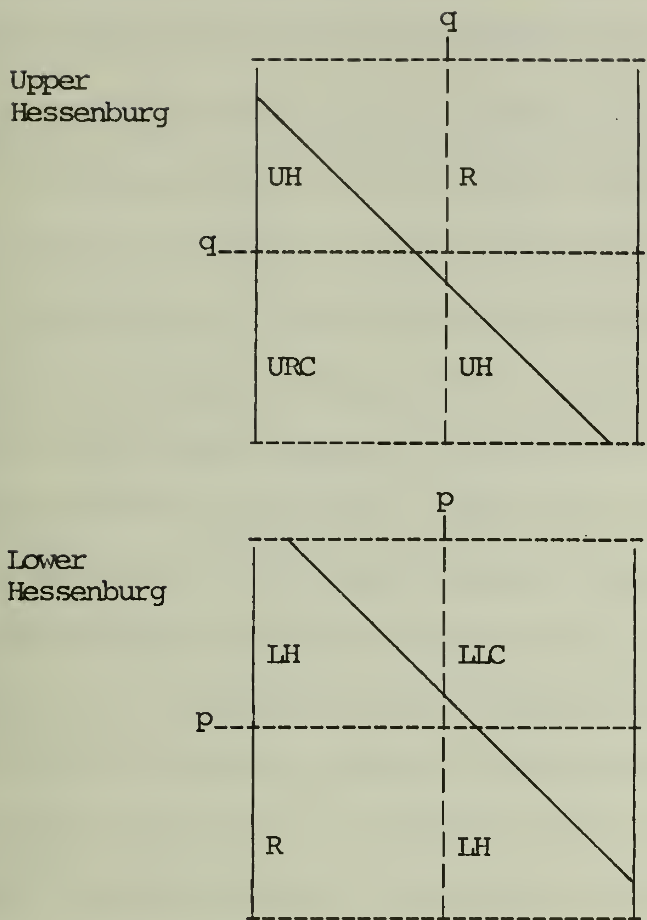


Figure 4. Partitioning of Hessenburg arrays

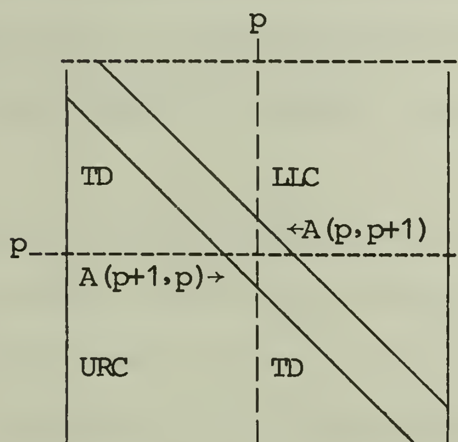


Figure 5. Partitioning of a tridiagonal array

The ACB control information for the non-corner subarrays is generated in the same manner as that generated for the existing array types. This includes use of the following formula for origin calculation:

$$\mathrm{LOC}(A(i+\delta i,j+\delta j)) = \omega + \delta R(i)\delta i + \delta D(\delta i(\delta i-1))/2 + \delta j$$

where $\omega = \mathrm{LOC}(A(1,1))$ and $i=j=1$. The values of δD and δR are copied from the corresponding fields in the parent array ACB. The subarray bounds are calculated from the parent array bounds and the partition lines in the usual manner.

The generation of the ACBs for the corner type arrays is easy because of the special form of the arrays. Due to the fact that a corner array consists of only one accessible element, the δR and δD fields are not needed and are set to zero. The origin field, ω, calculated by the above formula, points to this only non-zero element. The coordinates of this element with respect to the parent array are $(q+1,q)$ for URC and $(p,p+1)$ for LLC, where p is the partition line which bounds the array on the left (or bottom), and q is the partition line which bounds the array on the right (or top). The bounds, again, are a simple function of the parent array size and the partition lines.

The partitioning of subarrays of Hessenburg type is identical to the partitioning of the original Hessenburg array. The arbitrary partitioning of a corner array yields a corner subarray of the same type and various null subarrays.

The introduction of URC and LLC arrays is of importance in the partitioning of the existing tridiagonal type array. With this type, partitioning is also restricted to simultaneous row and column partition lines. In the case of one row and column partition line, the subarrays consist of two tridiagonal and one each of URC and LLC type. See Figure 5. In the past, the element in each of the corner arrays was unaccessible except by explicit

element subscripting, i.e., A(p+1,p) or A(p,p+1). With the implementation of the corner arrays, this unwanted restriction can be lifted.

2.4 Compiler Tables for Arithmetic and "Part of" Operations

The addition of the new array types, UH, LH, URC, and LLC more than double the size of the tables used to determine the array type of the sum or product of two matrices. See Figures 6 and 7. Of importance, though, are the starred entries in which the new array types allow more efficient use of storage in those cases where the result is of Hessenburg type. Because this array type was not available previously, a rectangular array type had to be used.

A feature of OL/2 allows the user to take a 'part of' an array, such as the upper triangular part of a square array. The π field of the PCB of the parent array, P, is used to point at the ACB of the 'part of' subarray, S, as shown in Figure 8. With the addition of Hessenburg array types, one must allow the user to take the 'part of' a Hessenburg array, and to take the upper or lower Hessenburg 'part of' other arrays under the existing limitation that all of the elements of the subarray must be accessible elements of the parent array. The rules for generating the bounds and origin control information of the subarray ACB are given in Figure 9. The δR and δD fields are copied from the corresponding fields in the parent array ACB.

+	URC	LLC	LH	UH	SLT	LT	TD	D	UT	SUT	R
URC	URC	R	R	UH	R	R	UH	UT	UT	SUT	R
LLC	R	LLC	LH	R	SLT	LT	LH	LT	R	R	R
LH	R	LH	LH	R	LH	LH	LH	LH	R	R	R
UH	UH	R	R	UH	R	R	UH	UH	UH	UH	R
SLT	R	SLT	LH	R	SLT	LT	LH*	LT	R	R	R
LT	R	LT	LH	R	LT	LT	LH*	LT	R	R	R
TD	UH	LH	LH	UH	LH*	LH*	TD	TD	UH*	UH*	R
D	UT	LT	LH	UH	LT	LT	TD	D	UT	UT	R
UT	UT	R	R	UH	R	R	UH*	UT	UT	UT	R
SUT	SUT	R	R	UH	R	R	UH*	UT	UT	SUT	R
R	R	R	R	R	R	R	R	R	R	R	R

Figure 6. Geometric type of sum of two matrices

×	URC	LLC	LH	UH	SLT	LT	TD	D	UT	SUT	R
URC	Φ	D	UT	SUT	UT	UT	SUT	URC	URC	Φ	UT
LLC	D	Φ	SLT	LT	Φ	SLT	SLT	LLC	LT	LT	LT
LH	UT	SLT	R	R	LT	LH	R	LH	R	R	R
UH	SUT	LT	R	R	R	R	R	UH	UH	UT	R
SLT	UT	Φ	LT	R	SLT	SLT	LT	SLT	R	R	R
LT	UT	SLT	LH	R	SLT	LT	LH*	LT	R	R	R
TD	SUT	SLT	R	R	LT	LH*	R	TD	UH*	UT	R
D	URC	LLC	LH	UH	SLT	LT	TD	D	UT	SUT	R
UT	URC	LT	R	UH	R	R	UH*	UT	UT	SUT	R
SUT	Φ	LT	R	UT	R	R	UT	SUT	SUT	SUT	R
R	UT	LT	R	R	R	R	R	R	R	R	R

where Φ is a null array

Figure 7. Geometric type of product of two matrices

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

ACB(P)

η = name	$\Delta = 2$
$\tau = R$	$\delta R = 5$
$\omega = 1$	$\delta D = 0$
$\lambda_1 = 1$	$\mu_1 = 5$
$\lambda_2 = 1$	$\mu_2 = 5$
ρ	

PCB(P)

π	
$R_0 = 0$	$C_0 = 0$
$R_1 = 5$	$C_1 = 5$
	$\nu = 0$
σ = null	

ACB(S)

η = name	$\Delta = 2$
$\tau = UT$	$\delta R = 5$
$\omega = 1$	$\delta D = 0$
$\lambda_1 = 1$	$\mu_1 = 5$
$\lambda_2 = 1$	$\mu_2 = 5$
ρ = null	

Figure 8. Data structure for UT part of rectangular array

$\tau(P)$ / $\tau(S)$	LH	UH	SLT	LT	D	SUT	UT	TD	R
LH	X	X	X	X	X	X	X	X	1,5
UH	X	X	X	X	X	X	X	X	1,5
SLT	2,6	X	X	2,4	X	X	X	X	1,4
LT	2,5	X	X	X	X	X	X	X	1,5
D	2,5	2,5	X	2,5	X	X	2,5	2,5	1,5
SUT	X	2,3	X	X	X	X	2,3	X	1,3
UT	X	2,5	X	X	X	X	X	X	1,5
TD	2,5	2,5	X	X	X	X	X	X	1,5
R	X	X	X	X	X	X	X	X	X

KEY	RULE	
1	$\mu_i(S) = \min(\mu_1(P), \mu_2(P))$	$i = 1,2$
2	$\mu_i(S) = \mu_i(P)$	$i = 1,2$
3	$\omega(S) = \omega(P) + 1$	
4	$\omega(S) = \omega(P) + \delta R(P)$	
5	$\omega(S) = \omega(P)$	
6	$\omega(S) = \omega(P) + 2$	
X	not allowed	

Figure 9. Rules for calculating $\mu_i(S)$ and $\omega(S)$

3. BAND FORM

3.1 Representation of Band Arrays

A band array may be described as an n×n array in which the subscripts of the theoretically non-zero elements satisfy the relation:

$$|i-j| \leq w-1 \quad \text{for } 1 \leq i,j \leq n,$$

where w is the width of the array, i.e., the number of non-zero elements in the first row. See Figure 10. The number of non-zero elements is w(2n-w+1)-n.

Other array types in OL/2 have a constant δD, i.e., a constant increase or decrease in the number of elements in adjacent rows. This constant is known at compile time from the array type. Band arrays, on the other hand, do not have this property. The values of δR(i) for row i, $1 \leq i \leq (n-1)$, go through the values:

$$w,\ w+1,\ \ldots,\ \overbrace{w+a,\ w+a,\ \ldots,\ w+a}^{b},\ w+a-1,\ \ldots,\ w$$

where both a and b are functions of the order and width of the array:

$$a = \max(\min(n-w,w-2),0) \quad \text{for } w \geq 2$$

$$b = \max(n-2w+3,2w-n-1) \quad \text{for } w \geq 2\ .$$

Clearly, the present OL/2 ACB structure is not sufficient to describe band arrays. How can band arrays be described? What ACB information is necessary? The answers to these questions are the goal of the following discussion.

Band arrays consist of three subarrays in each of which δD is constant. Let us define:

$$\alpha = \min(w,n-w)$$

$$\beta = \max(w,n-w)$$

$$
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1w} & & \\
a_{21} & a_{22} & & & \ddots & \\
\vdots & & & & & a_{n-w+1,n} \\
a_{w1} & & & & & \vdots \\
& \ddots & & & & \\
& & a_{n,n-w+1} & \cdots & & a_{nn}
\end{bmatrix}
$$

Figure 10. Band array

then

$$\delta D=1 \quad \text{for } 1 \leq \text{row} \leq \alpha$$

$$\delta D=0 \quad \text{for } \alpha < \text{row} \leq \beta$$

$$\delta D=-1 \quad \text{for } \beta < \text{row} < n.$$

See Figure 11. Given the row value, δD is known, and therefore $\delta R(i)$ is known. See Figure 12. Care must be taken, however, when crossing over from one subarray to another, and in the special cases when w=1 and w=n.

Of interest are the number of theoretically non-zero elements in each of these subarrays defined by δD. Let T1 be the subarray in which $\delta D=1$, T2 where $\delta D=0$, T3 where $\delta D=-1$. Let N(Ti) denote the number of elements in Ti. Then

$$N(T1) = \alpha(\alpha+2w-1)/2$$

$$N(T2) = (n-2\alpha)\min(2w-1,n)$$

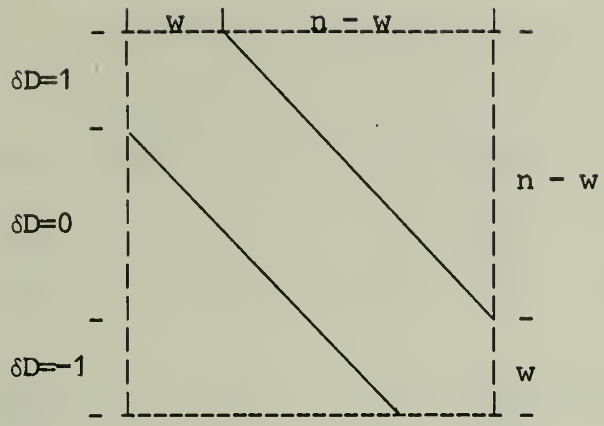
$$N(T3) = N(T1) \quad \text{by symmetry.}$$

Of fundamental importance is the answer to the question, "Given a band array of order n and width w, what is the sequential storage location of A(i,j)?" First, it must be determined whether A(i,j) is a stored element. The subscripts of stored elements satisfy the relation:

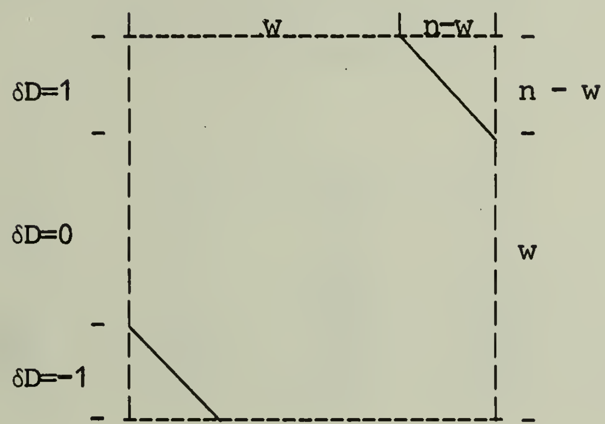
$$|i-j| \leq (w-1) \quad \text{for } 1 \leq i,j \leq n\ .$$

Let us assume that A(i,j) is a stored element. Then the location of A(i,j) is given by one of the following formulas depending on the value of i:

Case 1. $w < n - w$



Case 2. $w > n - w$



Case 3. $w = n - w$

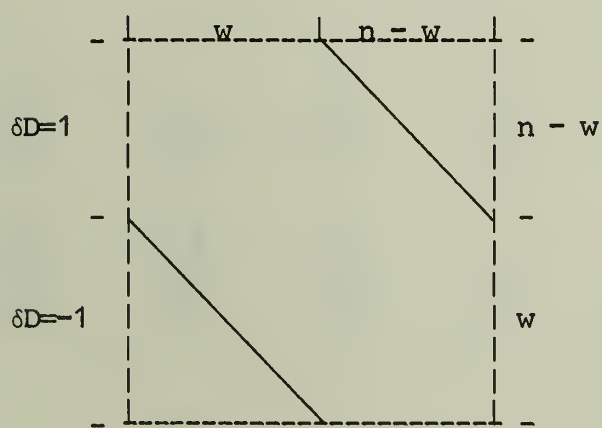


Figure 11. Subarrays of band array defined by δD

		Case 1. $w < n - w$	Case 2. $w > n - w$	Case 3. $w = n - w$
		$\delta R(1) = w$	$\delta R(1) = w$	$\delta R(1) = w$
		$\delta R(2) = w + 1$	$\delta R(2) = w + 1$	$\delta R(2) = w + 1$
$\delta D=1$	T1	$\vdots$	$\vdots$	$\vdots$
		$\delta R(w-1) = 2w - 2$	$\delta R(n-w-1) = n - 2$	$\delta R(w-1) = 2w - 2$
		$\delta R(w) = 2w - 2$	$\delta R(n-w) = n - 1$	$\delta R(w) = 2w - 2$
		$\delta R(w+1) = 2w - 2$	$\delta R(n-w+1) = n^{\dagger}$	
		$\delta R(w+2) = 2w - 2$	$\delta R(n-w+2) = n$	
$\delta D=0$	T2	$\vdots$	$\vdots$	empty
		$\delta R(n-w-1) = 2w - 2$	$\delta R(w-1) = n$	
		$\delta R(n-w) = 2w - 2$	$\delta R(w) = n - 1^{\dagger}$	
		$\delta R(n-w+1) = 2w - 2$	$\delta R(w+1) = n - 2$	$\delta R(w+1) = 2w - 2$
		$\delta R(n-w+2) = 2w - 3$	$\delta R(w+2) = n - 3$	$\delta R(w+2) = 2w - 3$
$\delta D=-1$	T3	$\vdots$	$\vdots$	$\vdots$
		$\delta R(n-1) = 2w - w$	$\delta R(n-1) = w$	$\delta R(n-1) = w$
		$\delta R(n)$ = not needed	$\delta R(n)$ = not needed	$\delta R(n)$ = not needed

† when $n-w+1 = w$, i.e. when T2 has only one row, use $n - 1$.

Figure 12. $\delta R(i)$ for band arrays

for $1 \leq i \leq \alpha$ (when $w=n$, no i will satisfy this relation),

$$LOC(A(i,j)) = LOC(A(1,1)) + (i-1)(i+2w-2)/2 + j - 1$$

for $\alpha < i \leq \beta$ (when $w=n-w$, no i will satisfy this relation),

$$LOC(A(i,j)) = LOC(A(1,1)) + N(T1) + (i-\alpha-1)(\min(2w-1,n)) + j - x$$

where $x=1$ for $\alpha=n-w$, and $x=i-\alpha+1$ for $\alpha=w$

for $\beta < i \leq n$ (when $w=n$, no i will satisfy this relation),

$$LOC(A(i,j)) = LOC(A(1,1)) + N(T1) + N(T2) + (i-\beta-1)(\min(2w-1,n-1)) - (i-\beta-2)(i-\beta-1)/2 + j - i + w - 1$$

where $LOC(A(1,1)) = 1$ for the original array.

Let us now assume that $A(i,j)$ is not a stored element. Then it is necessary for partitioning, as shown later, to find the smallest value of i' (or j') for which $A(i',j)$ (or $A(i,j')$) is stored. There are two cases (Figure 13):

for $i-j > (w-1)$

$1 \leq i \leq \alpha$ - not possible

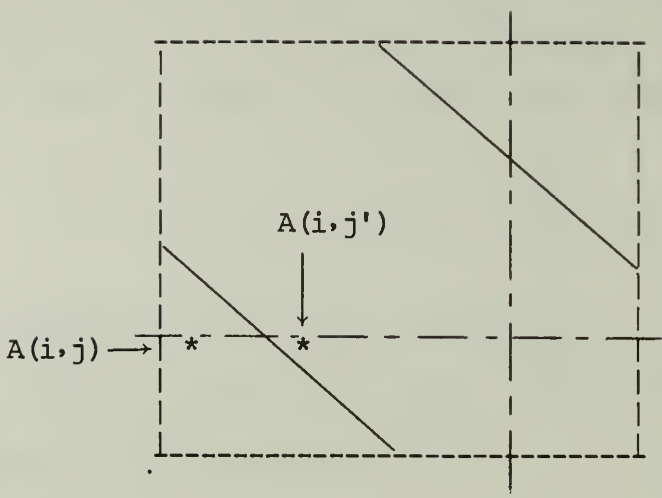
$$\left.\begin{array}{l} \alpha < i \leq \beta \\ \beta < i \leq n \end{array}\right\} \quad j' = i - w + 1$$

for $i-j < -(w-1)$

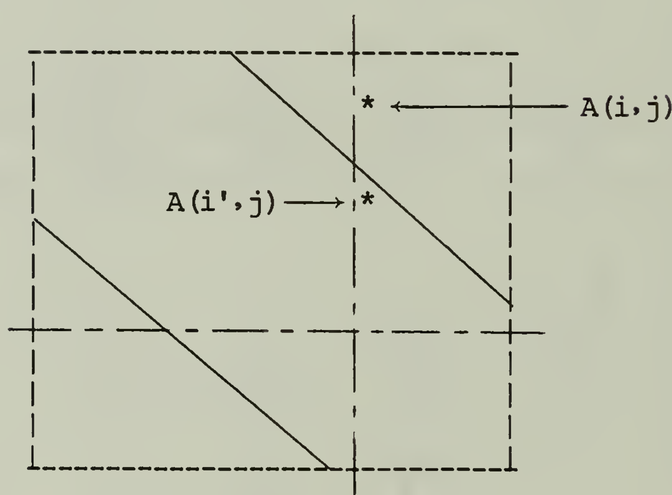
$$\left.\begin{array}{l} 1 \leq i \leq \alpha \\ \alpha < i \leq \beta \end{array}\right\} \quad i' = j - w + 1$$

$\beta < i \leq n$ - not possible.

The sequential location of the stored element $A(i',j)$ (or $A(i,j')$) can now be found by the previous formulas.



Case 1. $i - j > (w - 1)$



Case 2. $i - j < -(w - 1)$

Figure 13. Finding the nearest stored element in the same row or column

The previous discussion has exemplified the dependence of the representation of band arrays on n and w. Partitioning, as will again be shown, requires knowledge of absolute subscripts. From these considerations, the ACB structure for band arrays is proposed. See Figure 14. The η, Δ, τ, ω, λ_i, μ_i, and ρ fields have the same interpretation as the corresponding fields in existing ACBs. In place of the δR and δD fields are the w, s, n', and w' fields. The w field, along with a μ_i field, describe the size and shape of the array. The s, n', and w' fields describe the location of this array with respect to the original parent array (which may be itself) for the purpose of determining how δD varies in this array. All of this ACB information for the original array is available from the array declaration.

3.2 Partitioning of Band Arrays

The partitioning of band arrays is restricted, as is that for other non-rectangular arrays, to simultaneous identical row and column partition lines. Such partitioning gives rise to subarrays of three major types: band, null, and a new type, DB (diagonally bounded array).

An upper (or lower) DB array may be described as an n×m array in which all of the theoretically non-zero elements lie above (or below) one of n+m diagonal lines. Being a subarray of a band array, DB arrays inherit the problems of ACB representation, δD and δR, and origin calculation. Luckily, most of the information about band arrays can be carried over. The ACB structure for DB arrays is given in Figure 15. The format of the ACB is the same as that for band arrays with only slightly different interpretation. The η, Δ, τ, ω, λ_i, μ_i, and ρ fields serve their same purposes. The w and μ_i fields describe the size and shape of the array. Here, though, w is not the width

η	Δ
τ	ω
λ_1	μ_1
λ_2	μ_2
w	s
n'	w'
ρ	

where η - name of array
Δ - dimensionality
τ - geometric type
ω - origin
λ_i - lower bounds
μ_i - upper bounds
w - width
s - (i,j) coordinates of upper left corner
n' - order of oldest ancestor
w' - width of oldest ancestor
ρ - partition pointer

Figure 14. ACB for band arrays

η	Δ
τ	ω
λ_1	μ_1
λ_2	μ_2
w	s
n'	w'
ρ	

where η - name of array

Δ - dimensionality

τ - geometric type

ω - origin

λ_i - lower bounds

μ_i - upper bounds

w - boundary line

s - (i,j) coordinates of upper left corner

n' - order of oldest ancestor

w' - width of oldest ancestor

ρ - partition pointer

Figure 15. ACB for diagonally bounded (DB) arrays

as in band arrays, but an integer designating the boundary line. See Figure 16. The s, n', and w' fields serve, as in band ACBs, to determine δD and δR.

Most of the ACB information for subarrays is copied from the corresponding fields of the parent array ACB. For subarrays of band type, A<i,i>, this includes Δ, τ, n', w'. The μ_i and s values are simple functions of the partition lines and the corresponding parent field. The value of ω is LOC(A(s)) as computed by the formulas of the previous section. The width, w, is min(μ_i of subarray, w of parent array).

With the exception of the τ, ω, and w fields, the DB subarray ACB information is generated in the same manner as that for band subarrays. Given that s, the (i,j) coordinates (with respect to the original array) of the upper left corner of the DB subarray, has been found, τ is determined by the rules:

τ = lower DB (LDB) if $i < j$

τ = upper DB (UDB) if $i > j$

Next, it is ascertained whether s is a stored element. If it is, then ω can be found. For UDB, the value of w is i-i'-1, where i' is the largest number greater than i such that A(i',j) is a stored element. Similarly, for LDB, w=j'-j+1. If s is not a stored element, then, for LDB, that i' > i is found such that A(i',j) is a stored element. Then ω=LOC(A(i',j)) and w=i-i'. Similarly, for UDB, ω=LOC(A(i,j')) and w=j'-j. This procedure may be applied to any non-band subarray, null subarrays being discovered when the "nearest" stored element lies outside the bounds of the subarray.

The partitioning of subarrays of band type follows the same procedure as that for the original band array. For DB subarrays, where arbitrary row and column partitioning is allowable, all subarrays are assumed to be the

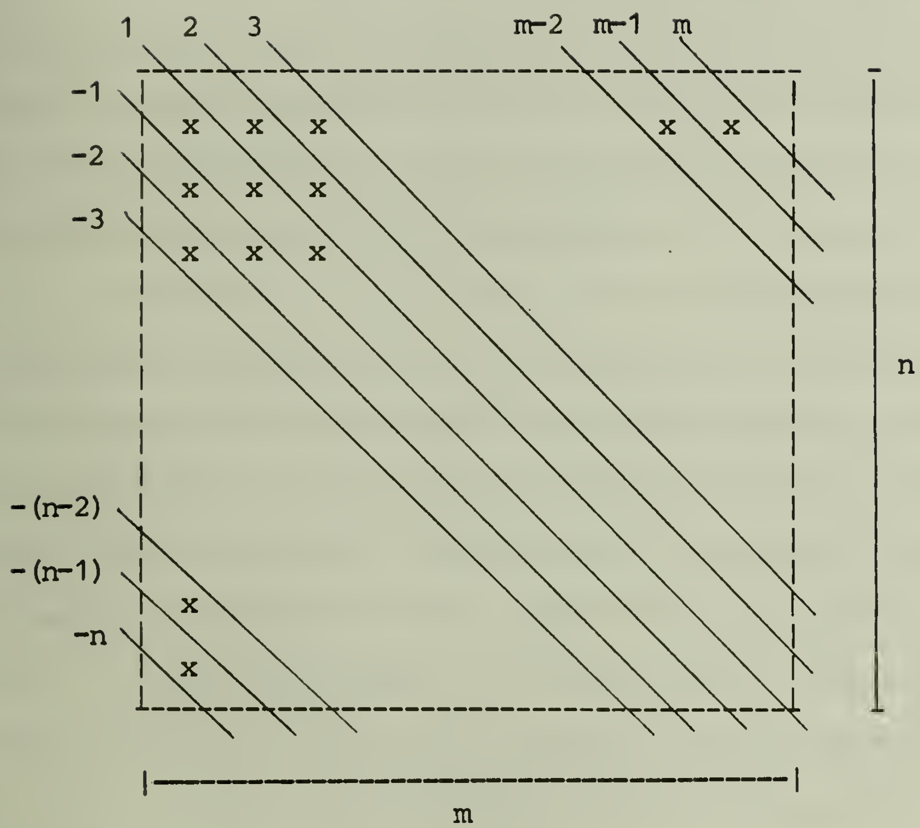


Figure 16. Values of w for boundary lines

same type as the parent with null subarrays being found by the same procedure as outlined above.

3.3 Discussion of Arithmetic and "Part of" Operations

The sum or product of a band or DB matrix, and another matrix results in a matrix whose type is dependent upon n and w (using either interpretation). In the present system, this necessitates defaulting the type of the result to rectangular. This action is in contradiction to the philosophy of not storing theoretically zero elements. It is not known whether functions can be found to determine the desired array type of the result from the type, n, and w of the operands.

The feature of taking "part of" has not been fully investigated for band arrays. The triangular "part of" a band array may be considered as a DB subarray and be represented as such. The band "part of" a band array can be implemented due to the n, w, n', and w' fields. This would include taking the diagonal and tridiagonal "part of" since both of these arrays are band arrays in the general sense. The band "part of" an array whose $\delta D=1$, 0 or -1 may be implemented by making the band part seem as if it were in a subarray of a larger band array where δD would have the same value.

4. ROTATE OPERATOR

4.1 Introduction

In an array language it is desirable to have many different geometric array types. This feature of a programming language allows great flexibility for the user in the approach of his problem. To have too few array types may make it necessary for the user to complicate or obscure the logic of his program by building "unusual" arrays from smaller available types. In the worst case, he may have to change algorithms. Such restrictions defeat the purpose of an array language.

One approach to the addition of new geometric array types is to define them explicitly. This is a major task involving finding ACB fields which adequately and efficiently represent the new types. Some of the arrays desired do not have a constant δD, and problems similar to those of band arrays arise. Origin calculation equations and representations of subarrays must be found. In effect, one must start anew for each array type to be added.

Another consideration is that of the size and complexity of the compiler tables which describe the operations of addition, multiplication, taking "part of", etc., on ordered pairs of array types. For example, the geometric type of the product of a tridiagonal matrix and a lower triangular matrix is required in the ACB of the temporary matrix which holds the result. For n array types, each of these tables is on the order of n^2. To double the number of array types, which is the intent, would increase the size of each table by a factor of four.

From the above considerations, it becomes clear that another approach should be considered. In the next section, we will consider the use of a result from group theory to simplify the problem of implementing additional geometric array types.

4.2 The D_4 Group

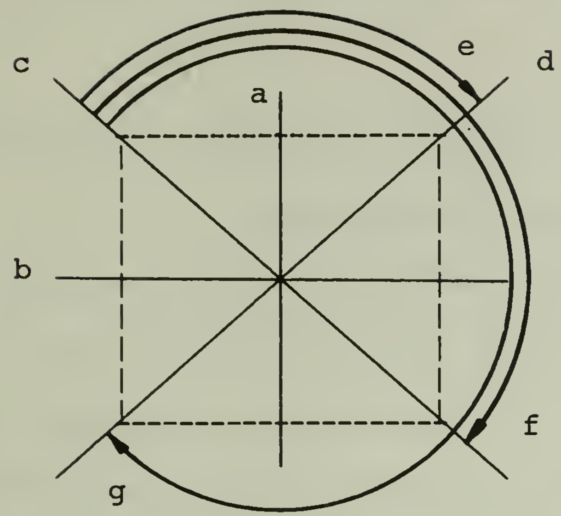
In this approach, the new array types are not defined explicitly, but are generated from existing ones. All of the information regarding array representation by ACB nodes and PCB nodes, however, remains the same.

The basis for this approach lies in the dihedral group of order four, D_4. This group has a geometric analog in the rotations of a square [4]. See Figure 17. The elements of D_4 consist of a, b, c, and d, rotations in space about axes of symmetry; e, f, and g, clockwise rotations of 90^0, 180^0, and 270^0, respectively, about the square's center in its plane; and h, the identity, no rotation. From the algebraic point of view, new array types could be obtained by operating on an available array type with an element or combination (product) of elements in D_4.

One need not implement all the elements of D_4 explicitly. Savings are made by implementing a subset of D_4 and generating the other elements by taking products of elements in this subset. While certain pairs of elements in D_4 will generate the whole group, i.e., {a,c}, the subset {a,b,c} was chosen for ease and simplicity of implementation of the remaining elements.

From the group table and the following equations, the elements of D_4 are shown to be generated by a, b, and c.

$$
\begin{aligned}
a &= a \\
b &= b \\
c &= c \\
d &= a \cdot b \cdot c \\
e &= b \cdot c \\
f &= a \cdot b \\
g &= a \cdot c \\
h &= a \cdot a
\end{aligned}
$$



	D_4	a	b	c	d	e	f	g	h
column reversal	a	h	f	g	e	d	b	c	a
row reversal	b	f	h	e	g	c	a	d	b
transpose	c	e	g	h	f	a	d	b	c
reverse transpose	d	g	e	f	h	b	c	a	d
90° rotation	e	c	d	b	a	f	g	h	e
180° rotation	f	b	a	d	c	g	h	e	f
270° rotation	g	d	c	a	b	h	e	f	g
identity - no rotation	h	a	b	c	d	e	f	g	h

Figure 17. Rotations of a square - the dihedral group, D_4

Since D_4 is not Abelian, these operators must be applied to an array from left to right:

$$(d)(A) = (a \cdot b \cdot c)(A) = (b \cdot c)(a(A)) = (c)(b(a(A)))$$

This fact is crucial in the implementation.

The set of array types resulting from operating on the current array types with elements of D_4 may be divided into two classes. One class consists of arrays of the same type as those available previously, but with the elements rearranged within the general shape. This class may be characterized by 1) symmetry with respect to the main diagonal, or 2) a boundary line parallel to the main diagonal. The other class consists of the new array types. These arrays are characterized by 1) symmetry with respect to the backward main diagonal, where the backward main diagonal is that line passing through the upper right and lower left corners of the array, or 2) a boundary line parallel to the backward main diagonal.

4.3 Implementation

4.3.1 Backward Diagonal Array

From the previous discussion, the construction of new array types involves only the implementation of the elements a, b, and c of D_4. Transposition, c, is already implemented. This leaves a and b, column and row reversal, respectively. These operations can be implemented with the use of a backward diagonal array. See Figure 18. A backward diagonal array is the identity array with the rows (or columns) reversed. Column reversal of a array, A, is accomplished in theory by post-multiplying A by a backward diagonal matrix of order m, where m is the number of columns in A. Row reversal,

$$
\begin{array}{cc}
 & n \\
n & \left[\begin{array}{cccccccc}
 & & & & & & & 1 \\
 & & & & & & 1 & \\
 & & 0 & & & \cdot & & \\
 & & & & \cdot & & & \\
 & & & \cdot & & & & \\
 & & \cdot & & & 0 & & \\
 & 1 & & & & & & \\
1 & & & & & & &
\end{array}\right]
\end{array}
$$

Figure 18. Backward diagonal array

similarly, is accomplished in theory by pre-multiplying A by a backward diagonal matrix of order n, where n is the number of rows in A. Because of the special form of backward diagonal arrays, these multiplications may be implemented with special techniques. Notice also that the transpose of a backward diagonal array is also a backward diagonal array. This fact leads to some simplification. An example for a 3×3 upper triangular array is given in Figure 19.

Thus, the syntax for the rotate operator could be (A, <integer>) where <integer> describes how the array A is to be rotated, i.e., which element of D_4 is to be applied to A. The compiler could then replace this term with the appropriate equivalent expression in terms of A, backward diagonal arrays, and the transpose operator. Since the dimensions of A are dynamic, the dimensions of the backward diagonal arrays cannot be set until execution time.

4.3.2 Setting of Bits in CALL Statement

The implementation of the transpose operator in the current version of OL/2 is done by means of flags. An addition, multiplication, or assignment operation results in a call to a subroutine to carry out the desired task. Some of the parameters in the argument list of the subroutine are flags indicating whether their associated arrays are to be transposed before operating with them. A simplified example is:

```
CALL MULT(A,0,B,1,C);
```

which forms $A \times B^T$ and assigns the result to C. The flag indicates how the elements of its corresponding array are to be accessed. The elements are not physically transposed. A flag value of 1 designates that element A(i,j) of the

$$\overset{A}{\begin{vmatrix}1&2&3\\&4&5\\&&6\end{vmatrix}} \times \overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} = \overset{RESULT}{\begin{vmatrix}3&2&1\\5&4&\\6&&\end{vmatrix}}$$

a
column reversal
A×BD

$$\overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} \times \overset{A}{\begin{vmatrix}1&2&3\\&4&5\\&&6\end{vmatrix}} = \begin{vmatrix}&&6\\&4&5\\1&2&3\end{vmatrix}$$

b
row reversal
BD×A

$$\overset{A'}{\begin{vmatrix}1&&\\2&4&\\3&5&6\end{vmatrix}} = \begin{vmatrix}1&&\\2&4&\\3&5&6\end{vmatrix}$$

c
transpose
A'

$$\overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} \times \overset{A'}{\begin{vmatrix}1&&\\2&4&\\3&5&6\end{vmatrix}} \times \overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} = \begin{vmatrix}6&5&3\\&4&2\\&&1\end{vmatrix}$$

d = a•b•c
reverse transpose
(BD×A×BD)' = BD'×A'×BD'
= BD×A'×BD

$$\overset{A'}{\begin{vmatrix}1&&\\2&4&\\3&5&6\end{vmatrix}} \times \overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} = \begin{vmatrix}&&1\\&4&2\\6&5&3\end{vmatrix}$$

e = b•c
90° rotation
(BD×A)' = A'×BD'
= A'×BD

$$\overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} \times \overset{A}{\begin{vmatrix}1&2&3\\&4&5\\&&6\end{vmatrix}} \times \overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} = \begin{vmatrix}6&&\\5&4&\\3&2&1\end{vmatrix}$$

f = a•b
180° rotation
BD×A×BD

$$\overset{BD}{\begin{vmatrix}&&1\\&1&\\1&&\end{vmatrix}} \times \overset{A'}{\begin{vmatrix}1&&\\2&4&\\3&5&6\end{vmatrix}} = \begin{vmatrix}3&5&6\\2&4&\\1&&\end{vmatrix}$$

g = a•c
270° rotation
(A×BD)' = BD'×A'
= BD×A'

Figure 19. Backward diagonal method applied to UT 3×3 array

transposed matrix is actually stored in location A(j,i) of the original array.

Both row reversal and column reversal can be implemented in a similar manner. Let there be two more flags associated with each array in a CALL statement. These flags are then used to indicate whether the array's rows or columns, or both, are to be reversed before operating with it. Again the elements would not be physically moved. For row reversal, element A(i,j) of the new array is actually located in A(n-i+1,j) of the original array. Similarly, for column reversal, A(i,j) is actually in A(i,m-j+1), where the original array is m×n. Combinations of these two flags and the transpose flag exhaust all the elements of D_4. Note again that because D_4 is not Abelian, a, b, and c must be used, when applicable, in precisely that order.

This second approach seems much more appealing. No multiplications need be performed. In the backward diagonal array method, the result of the rotation must be stored in a rectangular temporary matrix and any further operations must use the zeros which are generated, whereas in the second method no additional storage is required and computational efficiency is retained. Finally, because of the similarity of the second method to the already implemented transpose operator, little additional program logic is required for its implementation.

5. CONCLUSION

The preceding sections have shown the feasibility of adding Hessenburg and band arrays, and a rotate operator to the OL/2 language. Most, if not all, of the necessary information for the implementation of Hessenburg arrays has been given and such implementation should therefore be straightforward. The same optimism is felt for the realization of the rotate operator, due to its similarity with the already implemented transpose operator. The implementation of band arrays, on the other hand, is expected to present some problems, caused in most part by the new ACB representation. All in all, with the inclusion of these new features, the user of the OL/2 language will hopefully find a shorter and easier path to the solution of his problems.

LIST OF REFERENCES

[1] Phillips, J. R., "The Structure and Design Philosophy of OL/2--An Array Language--Part I: Language Overview," University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 544, December 1972.

[2] Phillips, J. R., "The Structure and Design Philosophy of OL/2--An Array Language--Part II: Algorithms for Dynamic Partitioning," University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 420, September 1971.

[3] Phillips, J. R., "Dynamic Partitioning for Array Languages," Communications of the ACM, Vol. 15, No. 12, pp. 1023-1032, December 1972.

[4] Wells, Mark B., Elements of Combinatorial Computing, Oxford, Pergamon Press, 1971.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-73-612	2.	3. Recipient's Accession No.
4. Title and Subtitle DATA STRUCTURES AND OPERATOR FOR NEW ARRAY TYPES IN OL/2			5. Report Date December 1973
			6.
7. Author(s) John Leonard Larson			8. Performing Organization Rept. No. UIUCDCS-R-73-612
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. US NSF-GJ-328
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C. 20550			13. Type of Report & Period Covered Master's Thesis
			14.

15. Supplementary Notes

16. Abstracts

This report is concerned with the addition of new array types to the OL/2 language. The necessary information for the implementation of Hessenburg arrays and their subarrays is given, including ACB representation and algorithms for partitioning. Modified data structures are proposed for describing band arrays and their subarrays, and new algorithms for partitioning are derived. Additional array types are realized by means of a rotate operator, which can generate new array types from the existing ones. Two possible implementations are suggested.

17. Key Words and Document Analysis. 17a. Descriptors

array control block
array language
array partitioning
band arrays
Hessenburg arrays
partition control block
rotate operator

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited	UNCLASSIFIED	42
	20. Security Class (This Page) UNCLASSIFIED	22. Price